Theoretical Computer Science 402 (2008) 172–189

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Integration of a security type system into a program logic☆

Reiner Hähnle [a,*], Jing Pan [b], Philipp Rümmer [a], Dennis Walter [c]

[a] Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, Sweden
[b] Department of Mathematics and Computer Science, Eindhoven University of Technology, Netherlands
[c] Deutsches Forschungszentrum für Künstliche Intelligenz, Bremen, Germany

ARTICLE INFO

Keywords:
Language-based security
Information-flow analysis
Dynamic logic
Security type system
Formal verification

ABSTRACT

Type systems and program logics are often thought to be at opposing ends of the spectrum of formal software analyses. In this paper we show that a flow-sensitive type system ensuring non-interference in a simple while-language can be expressed through specialised rules of a program logic. In our framework, the structure of non-interference proofs resembles the corresponding derivations in a state-of-the-art security type system, meaning that the algorithmic version of the type system can be used as a proof procedure for the logic. We argue that this is important for obtaining uniform proof certificates in a proof-carrying code framework. We discuss in which cases the interleaving of approximative and precise reasoning allows us to deal with delimited information release. Finally, we present ideas on how our results can be extended to encompass features of realistic programming languages such as Java.

1. Introduction

Formal verification of software properties has recently attracted a lot of interest. An important factor in this trend is the enormously increased need for secure applications, particularly in mobile environments. Confidentiality policies can often be expressed in terms of information flow properties. Existing approaches to verification of such properties mainly fall into two categories: the first are type-based security analyses ([1] gives an overview), whereas the second are deduction-based and employ program logics (e.g., [2–4]).

It is often noted that type-based analyses have a very logic-like character: a language for judgements is provided, a semantics that determines the set of *valid* judgments, and, finally, typing rules that approximate the semantics mechanically. Type systems typically trade off a precise model of the underlying programming language semantics in the judgments for automation and efficiency: many valid judgments are rejected. For program logics, the situation is complementary: calculi try to capture the semantics as precisely as possible and therefore have significantly higher complexity than type systems. Furthermore, due to the richer syntax of program logics (as compared to type-based judgments) the framework is more general and the same program logic can be used to express and reason about several different kinds of program properties.

The main contributions of this paper are as follows: we construct a calculus for a program logic that naturally simulates the rules of a flow-sensitive type system for secure information flow. We prove soundness of the program logic calculus with respect to the type system. The so obtained interpretation of the type system in dynamic logic yields increased precision

☆ This work was funded in part by a STINT institutional grant and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This article reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein.

* Corresponding author.
E-mail address: reiner@cs.chalmers.se (R. Hähnle).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.04.033

and opens up ways of expressing properties beyond pure non-interference. Concretely, we are able to prove the absence of exceptions in certain cases, and we can express delimited information release. Therefore, we can speak of the integration of a security type system into program logic.

A crucial benefit of the integration is that we obtain an automatic proof procedure for non-interference formulae: because of the similarity between the program logic calculus and the type rules, it is possible to uniformly translate type derivations to deduction proofs in the program logic. At the same time, certain advantages over the type system in terms of precision (Section 6) come for free without sacrificing automation.

The paper is organised as follows. In Section 2 we argue that a formal connection between type systems and program logics fits nicely into a verification strategy for advanced security policies of mobile JAVA programs based on proof-carrying-code (PCC). Section 3 introduces the terminology used in the rest of the paper. In Section 4 we define and discuss our program logic tailored to non-interference analysis. Section 5 describes an optimisation of the calculus that leads to significantly smaller proofs. Our ideas for increasing the precision of the calculus and for covering delimited information release are given in Section 6. Proofs of non-trivial lemmas and theorems are given in the Appendix; fully worked out proofs are contained in [5].

2. Integrating type systems and program logics

We claim that the integration of type systems and program logics is an important ingredient to make security policy checks scale up to mobile code written in modern industrial programming languages such as JAVA CARD. One reason is that there are very few implementations of type-based security analyses for such languages. The system Jif [6] is the only system we are aware of. On the other hand, there are several verification systems that cover large parts of JAVA with expressive program logics that allow to formulate a wide range of security properties [7–9]. Examples of such properties include absence of uncaught exceptions, well-formedness of atomic transactions, unwanted arithmetic overflow, and information leakage. The first three of these have been modelled in dynamic logic in [10]. We look at the last property in detail in the present paper. In addition, an integration of type systems and program logic as suggested here creates the possibility to design uniform logic-based certificates for proof-carrying code and it allows to exploit synergies between type-based and deduction-based reasoning. We elaborate the last two points in more detail.

Certificates for proof-carrying code. For the security infrastructure of mobile, ubiquitous computing it is essential that security policies can be enforced locally on the end-user device without requiring a secure internet connection to a trusted authentication authority. In the EU project MOBIUS[1] this infrastructure is based on the proof-carrying code (PCC) technology [11]. The basic idea of PCC is to provide a formal proof that a security policy holds for a given application, and then to hand down to the code consumer (end user) not only the application code, but also a certificate that allows to reconstruct the security proof locally with low overhead. Therefore, the end user device must run a proof checker, and, in a standard PCC architecture [11], also a verification condition generator, because certificates do not contain aspects of programs. The latter makes the approach unpractical for devices with limited resources. In addition, the security policies considered in MOBIUS [12] are substantially more complex than the safety policies originally envisioned in PCC. In foundational PCC [13] this is dispensed with at the price of including the formal semantics of the target language in the proof checker. The size of the resulting proof certificates makes this approach impractical so far.

In the case of an independently checked axiomatic semantics as used in the verification system employed in the present paper [9], it seems possible to arrive at a *trusted code base* that is small enough. This idea is pursued under the name of *logic-based* PCC (because the resulting certificates are logic-based) in MOBIUS. In the type-based version of PCC the trusted code base consists of a type checker instead of a proof checker. The integration of a type system for secure information flow into a program logic makes it possible to construct uniformly logic-based certificates, and no hybrid certificates need to be maintained. As a consequence, the PCC architecture is simplified and the trusted code base is significantly reduced. Efforts that go into similar directions in the sense that the expressivity of certificates is extended include Configurable PCC [14] and Temporal Logic PCC [15].

The design of efficient logic-based proof certificates is beyond the scope of this paper. It is a substantial task to which a whole Work Package within MOBIUS is devoted.

Synergies from combining type-based and deduction-based verification. The possibility to combine type-based and deduction-based reasoning in one framework leads to a number of synergies. In an integrated type- and deduction-based framework it is possible to increase the precision of the analysis dynamically on demand. Type systems ignore the values of variables. In a deduction framework, however, one can, e.g., prove that in the program "**if** (b) $y = x$; **if** $(\neg b)$ $z = y$ "; the variables z and x are independent, because the value of b always excludes the path through one of the conditionals. Note that it is not necessary to track the values of all variables to determine this: only the value of b matters in the example. More realistic examples are in Section 6.

[1] mobius.inria.fr/twiki/bin/view/Mobius.

$$\frac{}{p \vdash^{\mathrm{HS}} \nabla \{\ \} \nabla}\ \textsc{Skip}^{\mathrm{HS}}$$

$$\frac{\nabla \vdash E : t}{p \vdash^{\mathrm{HS}} \nabla \{\ v = E\ \} \nabla[v \mapsto p \sqcup t]}\ \textsc{Assign}^{\mathrm{HS}}$$

$$\frac{p \vdash^{\mathrm{HS}} \nabla \{\ \alpha_1\ \} \nabla' \qquad p \vdash^{\mathrm{HS}} \nabla' \{\ \alpha_2\ \} \nabla''}{p \vdash^{\mathrm{HS}} \nabla \{\ \alpha_1\ ;\ \alpha_2\ \} \nabla''}\ \textsc{Seq}^{\mathrm{HS}}$$

$$\frac{\nabla \vdash b : t \qquad p \sqcup t \vdash^{\mathrm{HS}} \nabla \{\ \alpha_i\ \} \nabla' \ \ (i = 1, 2)}{p \vdash^{\mathrm{HS}} \nabla \{\ \textbf{if}\ b\ \alpha_1\ \alpha_2\ \} \nabla'}\ \textsc{If}^{\mathrm{HS}}$$

$$\frac{\nabla \vdash b : t \qquad p \sqcup t \vdash^{\mathrm{HS}} \nabla \{\ \alpha\ \} \nabla}{p \vdash^{\mathrm{HS}} \nabla \{\ \textbf{while}\ b\ \alpha\ \} \nabla}\ \textsc{While}^{\mathrm{HS}}$$

$$\frac{p_1 \vdash^{\mathrm{HS}} \nabla_1 \{\ \alpha\ \} \nabla_1'}{p_2 \vdash^{\mathrm{HS}} \nabla_2 \{\ \alpha\ \} \nabla_2'}\ \textsc{Sub}^{\mathrm{HS}} \qquad \begin{array}{l} p_2 \sqsubseteq p_1, \\ \nabla_2 \sqsubseteq \nabla_1,\ \ \nabla_1' \sqsubseteq \nabla_2' \end{array}$$

Fig. 1. Hunt and Sands' flow-sensitive type system for information flow analysis.

A further opportunity offered by the integration of type-based analysis into an expressive logical framework is the formulation of additional security properties without the need for substantial changes in the underlying rule system or the deduction engine. To illustrate this point we show in Section 6 that it is possible to express delimited information release in our program logic.

3. Background and terminology

3.1. Non-interference analysis

Roughly speaking, a program has secure information flow if no knowledge about secret input data can be gained by observing the behaviour on public data of this program. Whether or not a program has secure information flow can hence only be decided according to a given security policy discriminating secret from public data. In our considerations we adopt the common model where all input and output channels are taken to be program variables. The semantic concept underlying secure information flow is then that of non-interference: nothing can be learned about a secret initially stored in variable `h` ("high"), by observing variable `l` ("low") after program execution, if the initial value of `h` *does not interfere with* the final value of `l`. Put differently, the final value of `l` must be *independent* of the initial value of `h`.

This non-interference property is commonly established via security type systems [1,16–18], where a program is deemed secure if it is typable according to some given policy. Type systems are used to perform flow-sensitive as well as flow-insensitive analyses. Flow-insensitive approaches (e.g. [17]) require every subprogram to be well-typed according to the *same* policy. Recent flow-sensitive analyses [16,18] allow the types of variables to change along the execution path, thereby providing more flexibility for the programmer. The program logic developed in this paper will be termination-insensitive, meaning that a security guarantee is only made about terminating runs of the program under consideration. The decision to make the program logic termination-insensitive comes from the fact that we wanted to model the type system of Hunt and Sands [16], which is termination-insensitive as well. It would be possible to design a termination-sensitive program logic along the lines stated in [4] using a total correctness modality.

Our starting point is the flow-sensitive type system of Hunt and Sands [16] which is depicted in Fig. 1. The type p represents the security level of the program counter and serves to eliminate indirect information flow. The remaining components of typing judgements are a program α and two typing functions $\nabla, \nabla' : \mathrm{PVar} \to \mathcal{L}$ mapping program variables to their respective pre- and post-types. The type system is parametric with respect to the choice of security types; it only requires them to form a (complete) lattice $\mathcal{L}$. In this paper, we will only consider the most general[2] lattice $\mathcal{P}(\mathrm{PVar})$. One may thus think of the type $\nabla(v)$ of a variable v as the set of all variables that v's value may depend on at a given point in the program. A judgement $p \vdash^{\mathrm{HS}} \nabla \{\ \alpha\ \} \nabla'$ states that in context p the program α transforms the typing (or dependency approximation) ∇ into ∇'. We note that rule $\textsc{Assign}^{\mathrm{HS}}$ gives the system its flow-sensitive character, stating that variable v's type is changed by an assignment $v = E$ to E's type as given by the pre-typing ∇ joined with the context type p. The type t of an expression E in a typing ∇ can simply be taken to be the join of the types $\nabla(v)$ of all free variables v occurring in E, which we denote by $\nabla \vdash E : t$. Joining with the context p is required to accommodate for leakage through the program context, as in the program "**if** (h) $\{l = 1\}$ $\{l = 0\}$)", where the initial value of h is revealed in the final value of l. A modification of the context p can be observed, e.g., in rule $\textsc{If}^{\mathrm{HS}}$, where the subderivation of the two branches of an if statement must be conducted in a context lifted by the type of the conditional.

[2] In the sense that any other type lattice is subsumed by it, see [16, Lemma 6.8].

3.2. Dynamic logic with updates

Following [4], the program logic that we investigate is a simplified version of dynamic logic (DL) for JavaCard [19]. The most notable difference to standard first-order dynamic logic for the simple while-language [20] is the presence of an explicit operator for simultaneous substitutions called (state) *updates* [21]. Updates are particularly useful when more complex programming languages (with arrays or object-oriented features) are considered. For our present purposes they serve to allow a direct relation between program logic and type systems.

A *signature* of DL is a tuple $(\Sigma, \mathrm{PVar}, \mathrm{LVar})$ consisting of a set Σ of *function symbols* with fixed, non-negative arity, a set PVar of *program variables* and of a countably infinite set LVar of *logical variables*. Σ, PVar, LVar are pairwise disjoint. Because some of our rules need to introduce fresh function symbols, we assume that Σ contains infinitely many symbols for each arity n. Further, we require that a distinguished nullary symbol $TRUE \in \Sigma$ exists. *Rigid terms* t_r, *ground terms* t_g, *terms* t,[3] *programs* α, *updates* U and *formulae* ϕ are then defined by the following grammar, where $f \in \Sigma$ ranges over functions, $x \in \mathrm{LVar}$ over logical variables and $v \in \mathrm{PVar}$ over program variables:

$$
\begin{array}{ll}
t_r ::= x \mid f(t_r, \ldots, t_r) & t_g ::= v \mid f(t_g, \ldots, t_g) \\
t ::= t_r \mid t_g \mid f(t, \ldots, t) \mid \{U\}\, t & U ::= \epsilon \mid v := t,\ U \\
\phi ::= \ \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \forall x.\, \phi \mid \exists x.\, \phi \mid \neg\phi \mid t = t \mid [\alpha]\,\phi \mid \{U\}\,\phi & \\
\alpha ::= \alpha\ ;\ \ldots\ ;\ \alpha \mid v = t_g \mid \textbf{if}\ t_g\ \alpha\ \alpha \mid \textbf{while}\ t_g\ \alpha. &
\end{array}
$$

For the whole paper, we assume a fixed signature $(\Sigma, \mathrm{PVar}, \mathrm{LVar})$ in which the set $\mathrm{PVar} = \{v_1, \ldots, v_n\}$ is finite, containing exactly those variables occurring in the program under investigation.

A *structure* is a pair $S = (D, I)$ consisting of a non-empty *universe* D and an *interpretation* I of function symbols, where $I(f) : D^n \rightarrow D$ if $f \in \Sigma$ has arity n. *Program variable assignments* and *variable assignments* are mappings $\delta : \mathrm{PVar} \rightarrow D$ and $\beta : \mathrm{LVar} \rightarrow D$. The space of all program variable assignments over the universe D is denoted by $PA^D = \mathrm{PVar} \rightarrow D$, and the corresponding flat domain by $PA^D_\perp = PA^D \cup \{\perp\}$, where $\delta \sqsubseteq \delta'$ iff $\delta = \perp$ or $\delta = \delta'$.

While-programs α are evaluated in structures and operate on program variable assignments. We use a standard denotational semantics for such programs

$$[\![\alpha]\!]^S : PA^D \rightarrow PA^D_\perp$$

and define, for instance, the meaning of a loop "**while** $b\ \alpha$" through

$$[\![\textbf{while}\ b\ \alpha]\!]^S =_{\mathrm{def}} \bigsqcup_i w_i, \quad w_i : PA^D \rightarrow PA^D_\perp$$

$$w_0(\delta) =_{\mathrm{def}} \perp,\ w_{i+1}(\delta) =_{\mathrm{def}} \begin{cases} (w_i)_\perp([\![\alpha]\!]^S(\delta)) & \text{for } val_{S,\delta}(b) = val_S(TRUE) \\ \delta & \text{otherwise} \end{cases}$$

where we make use of a 'bottom lifting': $(f)_\perp(x) = \mathit{if}\ (x = \perp)\ \mathit{then}\ \perp\ \mathit{else}\ f(x)$. Evaluation *val* of terms and formulae is defined below.

Likewise, updates are given a denotation as total operations on program variable assignments. The statements of an update are executed in parallel, where conflicting assignments to the same program variable are resolved in such a way that statements occurring syntactically later will override the effect of earlier statements:

$$[\![U]\!]^{S,\beta} : PA^D \rightarrow PA^D$$

$$[\![w_1 := t_1, \ldots, w_k := t_k]\!]^{S,\beta}(\delta) =_{\mathrm{def}} (\cdots((\delta[w_1 \mapsto val_{S,\beta,\delta}(t_1)])[w_2 \mapsto val_{S,\beta,\delta}(t_2)])\cdots)[w_k \mapsto val_{S,\beta,\delta}(t_k)]$$

where $(\delta[w \mapsto a])(v) = \mathit{if}\ (v = w)\ \mathit{then}\ a\ \mathit{else}\ \delta(v)$ are function updates.

Evaluation $val_{S,\beta,\delta}$ of terms and formulae is mostly defined as it is common for first-order predicate logic. Formulae are mapped into a Boolean domain, where tt stands for semantic truth. The cases for programs and updates are

$$val_{S,\beta,\delta}([\alpha]\,\phi) =_{\mathrm{def}} \begin{cases} val_{S,\beta,[\![\alpha]\!]^S(\delta)}(\phi) & \text{for } [\![\alpha]\!]^S(\delta) \neq \perp \\ \mathrm{tt} & \text{otherwise} \end{cases}$$

$$val_{S,\beta,\delta}(\{U\}\,\phi) =_{\mathrm{def}} val_{S,\beta,[\![U]\!]^{S,\beta}(\delta)}(\phi).$$

We interpret free logical variables $x \in \mathrm{LVar}$ existentially: a formula ϕ is *valid* iff for each structure $S = (D, I)$ and each program variable assignment $\delta \in PA^D$ there is a variable assignment $\beta : \mathrm{LVar} \rightarrow D$ such that $val_{S,\beta,\delta}(\phi) = \mathrm{tt}$. A sequent $\Gamma \vdash^{\mathrm{dl}} \Delta$ is called valid iff $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is valid. The valuation function $val_{S,\beta,\delta}$ is canonically extended from formulae to sequents. If the argument of $val_{S,\beta,\delta}$ does not depend on β or δ (for example, if no program or logical variables occur), then these indices can be dropped.

The set of unbound variables occurring in a term or a formula t is denoted by $\mathrm{vars}(t) \subseteq \mathrm{PVar} \cup \mathrm{LVar}$. For program variables $v \in \mathrm{PVar}$, this means that $v \in \mathrm{vars}(t)$ iff v turns up anywhere in t. For logical variables $x \in \mathrm{LVar}$, we define $x \in \mathrm{vars}(t)$ iff x occurs in t and is not in the scope of $\forall x$ or $\exists x$.

[3] Both rigid terms and ground terms are terms.

We note that the semantic notion of non-interference can easily be expressed in the formalism of dynamic logic: One possibility [4] is to express the variable independence property introduced above as follows. Assuming the set of program variables is $\mathrm{PVar} = \{v_1, \ldots, v_n\}$, then v_j only depends on $v_1, \ldots, v_i$ if variation of $v_{i+1}, \ldots, v_n$ does not affect the final value of v_j:

$$\forall u_1, \ldots, u_i.\ \exists r.\ \forall u_{i+1}, \ldots, u_n.\ \{ v_i := u_i \}_{1 \le i \le n}\, [\alpha]\, (v_j = r). \tag{1}$$

The particular use of updates in this formula is a standard trick to quantify over program variables which is not allowed directly: in order to quantify over all values that a program variable v occurring in a formula ϕ can assume, we introduce a fresh logical variable u and quantify over the latter. In the following we use quantification over program variables as a shorthand, writing $\dot{\forall} v.\ \phi$ for $\forall u.\ \{ v := u \}\ \phi$. One central result of this paper is that simple, easily automated proofs of formulae such as (1) are viable in at least those cases where a corresponding derivation in the type system of Hunt and Sands exists.

4. Interpreting the type system in dynamic logic

We now present a calculus for dynamic logic in which the rules involving program statements employ abstraction instead of precise evaluation. The calculus facilitates automatic proofs of secure information flow. In particular, when proving loops the burden of finding invariants is reduced to the task of providing a dependency approximation between program variables. There is a close correspondence to the type system of [16] (Fig. 1). Intuitively, state updates in the DL calculus resemble security typings in the type system: updates arising during a proof will essentially take the form $\{v := f(\ldots vars \ldots)\}$, where the *vars* form the type of v in a corresponding derivation in the type system. To put our observation on a formal basis, we prove the soundness of the calculus and show that every derivation in the type system has a corresponding proof in our calculus.

4.1. The Abstraction-based calculus

We introduce *extended type environments* as pairs (∇, I) consisting of a typing function $\nabla : \mathrm{PVar} \to \mathcal{P}(\mathrm{PVar})$ and an *invariance set* $I \subseteq \mathrm{PVar}$. The latter is used to indicate those variables whose values do not change after execution of the program. We write ∇_v for the syntactic sequence of variables $w_1, \ldots, w_k$ with arbitrary ordering when $\nabla(v) = \{w_1, \ldots, w_k\}$ and ∇_v^C for a sequence of all variables *not* in $\nabla(v)$. Ultimately, we want to prove non-interference properties of the form

$$\{ \alpha \} \Downarrow (\nabla, I) \ \equiv_{\mathrm{def}} \ \bigwedge_{v \in \mathrm{PVar}} \begin{cases} \dot{\forall} v_1 \cdots v_n.\ \forall u.\ \{ v := u \}[\alpha]\, v = u, & v \in I \\ \dot{\forall} \nabla_v.\ \exists r.\ \dot{\forall} \nabla_v^C.\ [\alpha]\, v = r, & v \notin I \end{cases} \tag{2}$$

where we assume $\mathrm{PVar} = \{v_1, \ldots, v_n\}$. Validity of a judgment $\{ \alpha \} \Downarrow (\nabla, I)$ ensures that all variables in the invariance set I remain unchanged after execution of the program α (upper part), and that any non-invariant variable v depends at most on variables in $\nabla(v)$ (lower part).

The invariance set I corresponds to the context p that turns up in judgments $p \vdash^{\mathrm{HS}} \nabla \{ \alpha \} \nabla'$: while the type system ensures that p is a lower bound of the post-type $\nabla'(v)$ of variables v assigned in α, the set I can be used to ensure that variables with low post-type are not assigned (or, more precisely, not changed). The equivalence is formally stated in Lemma 2.

During the proof that the type system is faithfully interpreted in the program logic we are going to abstract program statements "**while** $b\ \alpha$" and "**if** $b\ \alpha_1\ \alpha_2$" into updates that model the effect of these statements. In this way we avoid having to split the proof in the program logic for the two branches of an if-statement, or having to find an invariant for a while-loop. Extended type environments capture the essence of updates that model the effect of complex statements, i.e. the arguments of the abstraction functions and the unmodified variables. We require the following extended type environments which are translated into updates:

$$\mathrm{upd}(\nabla, I) =_{\mathrm{def}} \{v := f_v(\nabla_v)\}_{v \in \mathrm{PVar} \setminus I}$$
$$\mathrm{ifUpd}(b, \nabla, I) =_{\mathrm{def}} \{v := f_v(b, \nabla_v)\}_{v \in \mathrm{PVar} \setminus I.}$$

The above updates assign to each v not in the invariance set I a *fresh* function symbol f_v whose arguments are exactly the variables given by the type $\nabla(v)$. In a program "**if** $b\ \alpha_1\ \alpha_2$" the final state may depend on the branch condition b, so the translation ifUpd 'injects' the condition into the update. This is the analogue of the context lifting present in $\mathrm{IF}^{\mathrm{HS}}$. For the while-rule, we transform the loop body into a conditional, so that we must handle the context lifting only in the if-rule.

Figs. 2 and 3 contain the rules for a sequent calculus. We have only included those propositional and first-order rules (the first rules of Fig. 2) that are necessary for proving the results in this section; more rules are required to make the calculus usable in practice (see, for instance, [22,9]). Some of the rules introduce *fresh* symbols, by which we mean symbols that have not yet occurred in the proof. The calculus uses free logical variables $X \in \mathrm{LVar}$ (EX-RIGHT$^{\mathrm{dl}}$) and unification (CLOSE-EQ$^{\mathrm{dl}}$) for handling existential quantification. The notation $[\, s \equiv t \,]$ in the latter rule expresses that the most general unifier of the terms s and t is applied to *the whole proof tree*. Because free variables can also occur in the scope of updates or the box modal operator, this is only correct if the unifier is *rigid* and maps all variables to rigid terms (not containing program variables). Skolemisation (ALL-RIGHT$^{\mathrm{dl}}$) has to collect the free variables that occur in a quantified formula to ensure soundness. By

$$\frac{\Gamma \vdash^{\mathrm{dl}} \phi, \Delta \quad \Gamma \vdash^{\mathrm{dl}} \psi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \phi \wedge \psi, \Delta}\ \textsc{and-right}^{\mathrm{dl}} \qquad \frac{\Gamma \vdash^{\mathrm{dl}} \phi, \psi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \phi \vee \psi, \Delta}\ \textsc{or-right}^{\mathrm{dl}}$$

$$\frac{\Gamma, \phi \vdash^{\mathrm{dl}} \psi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \phi \to \psi, \Delta}\ \textsc{imp-right}^{\mathrm{dl}} \qquad \frac{\Gamma, \phi \vdash^{\mathrm{dl}} \Delta}{\Gamma \vdash^{\mathrm{dl}} \neg\phi, \Delta}\ \textsc{not-right}^{\mathrm{dl}}$$

$$\frac{*}{\Gamma, \phi \vdash^{\mathrm{dl}} \phi, \Delta}\ \textsc{close}^{\mathrm{dl}}$$

Analogous: $\textsc{and-left}^{\mathrm{dl}}$, $\textsc{or-left}^{\mathrm{dl}}$, $\textsc{imp-left}^{\mathrm{dl}}$, $\textsc{not-left}^{\mathrm{dl}}$

$$\frac{\Gamma \vdash^{\mathrm{dl}} \phi[x/f(X_1, \ldots, X_n)], \Delta}{\Gamma \vdash^{\mathrm{dl}} \forall x.\, \phi, \Delta}\ \textsc{all-right}^{\mathrm{dl}} \qquad \begin{array}{l}\{X_1, \ldots, X_n\} = \\ \quad \mathrm{vars}(\phi) \cap \mathrm{LVar}\backslash\{x\}, \\ f \text{ fresh}\end{array}$$

$$\frac{\Gamma \vdash^{\mathrm{dl}} \phi[x/X], \exists x.\, \phi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \exists x.\, \phi, \Delta}\ \textsc{ex-right}^{\mathrm{dl}} \qquad X \text{ fresh}$$

Analogous: $\textsc{all-left}^{\mathrm{dl}}$, $\textsc{ex-left}^{\mathrm{dl}}$

$$\frac{[\, s \stackrel{*}{\equiv} t \,]}{\Gamma \vdash^{\mathrm{dl}} s = t, \Delta}\ \textsc{close-eq}^{\mathrm{dl}} \qquad \begin{array}{l}s, t \text{ unifiable} \\ \text{(with rigid unifier)}\end{array}$$

$$\frac{(\Gamma \vdash^{\mathrm{dl}} \Delta)[x/f(\mathrm{vars}(t))]}{(\Gamma \vdash^{\mathrm{dl}} \Delta)[x/t]}\ \textsc{abstract}^{\mathrm{dl}} \qquad f \text{ fresh}$$

$$\frac{\Gamma \vdash^{\mathrm{dl}} \{U\}\phi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \{U\}[\,]\phi, \Delta}\ \textsc{Skip}^{\mathrm{dl}} \qquad \frac{\Gamma \vdash^{\mathrm{dl}} \{U\}\{v := E\}[\ldots]\phi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \{U\}[v = E;\ \ldots]\phi, \Delta}\ \textsc{Assign}^{\mathrm{dl}}$$

$$\frac{\begin{array}{c}\vdash^{\mathrm{dl}} \{\,\alpha_i\,\} \Downarrow (\nabla, I) \quad (i = 1, 2) \\ \Gamma \vdash^{\mathrm{dl}} \{U\}\{\mathrm{ifUpd}(b, \nabla, I)\}[\ldots]\phi, \Delta\end{array}}{\Gamma \vdash^{\mathrm{dl}} \{U\}[\mathbf{if}\ b\ \alpha_1\ \alpha_2;\ \ldots]\phi, \Delta}\ \textsc{If}^{\mathrm{dl}}$$

$$\frac{\begin{array}{c}\vdash^{\mathrm{dl}} \{\,\mathbf{if}\ b\ \alpha\ \{\}\,\} \Downarrow (\gamma^*_\nabla(\nabla), I) \\ \Gamma \vdash^{\mathrm{dl}} \{U\}\{\mathrm{upd}(\nabla, I)\}[\ldots]\phi, \Delta\end{array}}{\Gamma \vdash^{\mathrm{dl}} \{U\}[\mathbf{while}\ b\ \alpha;\ \ldots]\phi, \Delta}\ \textsc{While}^{\mathrm{dl}} \qquad v \in \nabla(v) \text{ for all } v \in \mathrm{PVar}$$

Fig. 2. A dynamic logic calculus for information flow security. In the last four rules the update $\{U\}$ can also be empty and disappear.

$$\begin{array}{rcll}
\{w_1 := t_1, \ldots, w_k := t_k\}\, w_i & \to^{\mathrm{dl}} & t_i & \text{if } w_j \neq w_i \text{ for } i < j \leq k \\
\{w_1 := t_1, \ldots, w_k := t_k\}\, t & \to^{\mathrm{dl}} & t & \text{if } w_1, \ldots, w_k \notin \mathrm{vars}(t) \\
\{U\}\, f(t_1, \ldots, t_n) & \to^{\mathrm{dl}} & f(\{U\}\, t_1, \ldots, \{U\}\, t_n) & \\
\{U\}\, (t_1 = t_2) & \to^{\mathrm{dl}} & \{U\}\, t_1 = \{U\}\, t_2 & \\
\{U\}\, \neg\phi & \to^{\mathrm{dl}} & \neg\{U\}\, \phi & \\
\{U\}\, (\phi_1 * \phi_2) & \to^{\mathrm{dl}} & \{U\}\, \phi_1 * \{U\}\, \phi_2 & \text{for } * \in \{\vee, \wedge\}
\end{array}$$

$$\begin{array}{rcl}
\{U\}\, \{w_1 := t_1, \ldots, w_k := t_k\}\, \phi & & \\
\qquad \to^{\mathrm{dl}} & \{U,\ w_1 := \{U\}\, t_1, \ldots, w_k := \{U\}\, t_k\}\, \phi &
\end{array}$$

$$\frac{s \stackrel{*}{\to}^{\mathrm{dl}} t \qquad \Gamma, \phi[t] \vdash^{\mathrm{dl}} \Delta}{\Gamma, \phi[s] \vdash^{\mathrm{dl}} \Delta}\ \stackrel{*}{\to}^{\mathrm{dl}} \qquad \frac{s \stackrel{*}{\to}^{\mathrm{dl}} t \qquad \Gamma \vdash^{\mathrm{dl}} \phi[t], \Delta}{\Gamma \vdash^{\mathrm{dl}} \phi[s], \Delta}\ \stackrel{*}{\to}^{\mathrm{dl}}$$

Fig. 3. Application rules for updates in dynamic logic, as far as they are required for Lemma 6. Further application and simplification rules are necessary in general.

definition of the non-interference properties (2) and by the design of the rules of the dynamic logic calculus it is sufficient to define update rules for terms, quantifier-free formulae, and other updates. Such rules can be used at any point in a proof to simplify expressions containing updates.

Rule Abstract^{dl} can be used to normalise terms occurring in updates to the form $f(\ldots vars \ldots)$. In rules If^{dl} and While^{dl} the second premiss represents the actual abstraction of the program statement for a suitably chosen typing ∇ and invariance set I. This abstraction is justified through the first premiss in terms of another non-interference proof obligation. The concretisation operator γ^* (cf. [16]) of rule While^{dl} is generally defined as

$$\gamma^*_{\nabla_1}(\nabla_2)(x) \;=_{\text{def}}\; \{y \in \text{PVar} \mid \nabla_1(y) \subseteq \nabla_2(x)\} \quad (x \in \text{PVar}). \tag{3}$$

Together with the side condition that for all v we require $v \in \nabla(v)$, a closure property on dependencies is ensured; $w \in \gamma^*_\nabla(\nabla)(v)$ implies $\gamma^*_\nabla(\nabla)(w) \subseteq \gamma^*_\nabla(\nabla)(v)$; if a variable depends on another, the latter's dependencies are included in the former's. This accounts for the fact that the loop body can be executed more than once, which in general causes transitive dependencies.

Function arguments ensure soundness. A recurring proof obligation in a non-interference proof is a statement of the form $\dot\forall \nabla_v.\ \exists r.\ \dot\forall \nabla_v^C.\ [\alpha]\, v = r$, see (2). To prove this statement without abstraction requires to find a function of the variables ∇_v that yields the value of v under α for every given pre-state. In other words, one must find the strongest post-condition w.r.t. v's value. In a logic-based view this function corresponds to a term substituted for the existentially quantified variable r in which the ∇_v^C do not occur. In a unification-based calculus the occurs check will let all those proofs fail where an actual information flow from ∇_v^C to v takes place. The purpose of the arguments in the abstraction functions f_v that model the effect of complex statements is exactly to retain this crucial property in the abstract version of the calculus. We must make sure that an abstraction function f_v that approximates the effect of α on v has at least those variables as arguments that occur in the term representing the final value of v after execution of α.

Theorem 1 (*Soundness*). *The rules of the DL calculus given in Figs. 2 and 3 are sound: the root of a closed proof tree is a valid sequent.*

4.2. Simulating type derivations in the DL calculus

In order to show faithful interpretation of the type system in the program logic, we first put the connection between invariance sets and contexts on solid ground. It suffices to approximate the invariance of variables v with the requirement that v must not occur as left-hand side of assignments ($Lhs(\alpha)$ is the set of all left-hand sides of assignments in α).

Lemma 2. *In the type system of* [16] (Fig. 1) *the following equivalence holds:*

$$p \vdash^{\text{HS}} \nabla\ \{\ \alpha\ \}\ \nabla' \quad \textit{iff} \quad \bot \vdash^{\text{HS}} \nabla\ \{\ \alpha\ \}\ \nabla' \ \textit{ and f.a. } v \in Lhs(\alpha):\ p \sqsubseteq \nabla'(v).$$

Furthermore, we can normalise type derivations thanks to the Canonical Derivations Lemma of [16]. The crucial ingredient is the concretisation operator γ^* defined in (3).

Lemma 3 (*Canonical Derivations*).

$$\bot \vdash^{\text{HS}} \nabla\ \{\ \alpha\ \}\ \nabla' \quad \textit{iff} \quad \bot \vdash^{\text{HS}} \Delta_0\ \{\ \alpha\ \}\ \gamma^*_\nabla(\nabla') \quad \textit{where } \Delta_0 = \lambda x.\{x\}.$$

For brevity, we must refer to Hunt and Sands' paper for details, but in the setting at hand one can intuitively take Lemma 3 as stating that any typing judgment can also be understood as a dependency judgment: the typing on the left-hand side is equivalent to the statement that the final value of x *may depend on* the initial value of y only if y appears in the post-type, or dependence set, $\gamma^*_\nabla(\nabla')(x)$.

The type system of Fig. 4 only mentions judgments with a pre-type Δ_0 as depicted on the right-hand side of the equivalence in Lemma 3. Further, the context p has been replaced by equivalent side conditions (Lemma 2), and rule Seq$^{\text{HS}}$ is built into the other rules, i.e., the rules always work on the initial statement of a program. Likewise, rule Sub$^{\text{HS}}$ has been integrated in Skip$^{\text{cf}}$ and While$^{\text{cf}}$. The type system is equivalent to Hunt and Sands' system (Fig. 1):

Lemma 4.

$$\bot \vdash^{\text{HS}} \Delta_0\ \{\ \alpha\ \}\ \nabla \quad \textit{if and only if} \quad \vdash^{\text{cf}} \Delta_0\ \{\ \alpha\ \}\ \nabla.$$

The proof proceeds in multiple steps by devising intermediate type systems, each of which adds a modification towards the system in Fig. 4 and which is equivalent to Hunt and Sands' system. (The step from $\vdash^{\text{cf}}$ to $\vdash^{\text{dl}}$ is done in Lemma 6.)

Obviously, due to the approximative character of If^{dl} and While^{dl} (and the lack of arithmetic), our DL calculus is not (relatively) complete in the sense of [20]. For the particular judgements $\{\ \alpha\ \} \Downarrow (\nabla, I)$ the calculus is, however, not more incomplete than the type system of Fig. 1 — every typable program can also be proven secure using the DL calculus.[4]

[4] The converse of Theorem 5 does not hold. In the basic version of the calculus of Fig. 2, untypable programs like "**if** (h) $\{h = l; l = h\}$ $\{l = l\}$" can be proven secure. Section 6 discusses how the precision of the DL calculus can be further augmented.

$$\frac{}{\vdash^{\mathrm{cf}} \Delta_0 \{\ \}\ \nabla}\ \textsc{Skip}^{\mathrm{cf}} \qquad v \in \nabla(v) \text{ for all } v \in \mathrm{PVar}$$

$$\frac{\Delta_0 \vdash E : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \{ \ldots \}\ \gamma^*_{\Delta_0[v \mapsto t]}(\nabla)}{\vdash^{\mathrm{cf}} \Delta_0 \{\ v = E\ ;\ \ldots \}\ \nabla}\ \textsc{Assign}^{\mathrm{cf}}$$

$$\frac{\Delta_0 \vdash b : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \{ \ldots \}\ \gamma^*_{\nabla}(\nabla') \qquad \vdash^{\mathrm{cf}} \Delta_0 \{\ \alpha_i\ \}\ \nabla\ \ (i = 1, 2)}{\vdash^{\mathrm{cf}} \Delta_0 \{\ \mathbf{if}\ b\ \alpha_1\ \alpha_2\ ;\ \ldots \}\ \nabla'}\ \textsc{If}^{\mathrm{cf}} \qquad \begin{array}{l}\text{f.a. } v \in \mathit{Lhs}(\alpha_1).\ t \sqsubseteq \nabla(v) \\ \text{f.a. } v \in \mathit{Lhs}(\alpha_2).\ t \sqsubseteq \nabla(v)\end{array}$$

$$\frac{\Delta_0 \vdash b : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \{ \ldots \}\ \gamma^*_{\nabla}(\nabla') \qquad \vdash^{\mathrm{cf}} \Delta_0 \{\ \alpha\ \}\ \gamma^*_{\nabla}(\nabla)}{\vdash^{\mathrm{cf}} \Delta_0 \{\ \mathbf{while}\ b\ \alpha\ ;\ \ldots \}\ \nabla'}\ \textsc{While}^{\mathrm{cf}} \qquad \begin{array}{l}v \in \nabla(v) \text{ for all } v \in \mathrm{PVar} \\ \text{f.a. } v \in \mathit{Lhs}(\alpha). \\ \quad t \sqsubseteq \gamma^*_{\nabla}(\nabla)(v)\end{array}$$

Fig. 4. Intermediate flow-sensitive type system for information flow analysis.

Theorem 5.

$$\bot \vdash^{\mathrm{HS}} \Delta_0 \{\ \alpha\ \}\ \nabla \quad \textit{implies} \quad \vdash^{\mathrm{dl}} \{\ \alpha\ \} \Downarrow (\nabla, \emptyset).$$

The proof of the theorem is constructive: a method for translating type derivations into DL proofs is given. The existence of this translation mapping shows that proving in the DL calculus is, in principle, not more difficult than typing programs using the system of Fig. 1.

The first part of the translation is accomplished by Lemma 4 which covers structural differences between type derivations and DL proofs. Applications of the rules of Fig. 4 can then almost directly be replaced with the corresponding rules of the DL calculus:

Lemma 6.

$$\vdash^{\mathrm{cf}} \Delta_0 \{\ \alpha\ \}\ \nabla \quad \textit{implies} \quad \vdash^{\mathrm{dl}} \{\ \alpha\ \} \Downarrow (\nabla, \emptyset).$$

5. Reducing the size of proofs

Type derivations in the system cf of Fig. 4 grow linearly with the program size: there is always a one-to-one correspondence between the statements of a verified program α and the nodes labelled with $\textsc{Assign}^{\mathrm{cf}}$, $\textsc{If}^{\mathrm{cf}}$, or $\textsc{While}^{\mathrm{cf}}$ in a type derivation for α. This is a property that is lost in the DL calculus from Fig. 2, in which the size of a proof tree can grow exponentially in the number of if- and while-statements of a program. The reason lies in the rules $\textsc{If}^{\mathrm{dl}}$ and $\textsc{While}^{\mathrm{dl}}$, which split over the complete set PVar of program variables: recall from (2) that a non-interference statement $\{\ \alpha\ \} \Downarrow (\nabla, I)$ contains a conjunct for each program variable.

This complexity is not inherent to our calculus, but merely a consequence of the naive encoding (2) of type environments in formulae. Instead of inspecting the variables of a program one by one for independence according to their post-type, they can also be checked all at once, as it is done by the type system. Such an optimised encoding that avoids exponential blow-up is achieved by factoring out the common parts in the conjuncts of (2), expressing the differing parts with the help of conditional terms *if* ϕ *then* s *else* t. The resulting formulae can be proven in such a way that each statement of a program is handled by exactly one application of the rules $\textsc{Assign}^{\mathrm{dl}}$, $\textsc{If}^{\mathrm{dl}}$, or $\textsc{While}^{\mathrm{dl}}$ (provided that the formula is provable at all). However, proofs still grow quadratically in the number of program variables, which is natural considering that a type environment ∇ in our case is a mapping $\mathrm{PVar} \to \mathcal{P}(\mathrm{PVar})$.

As the selection of the program variable to be checked for independence is in the optimised encoding done on the object level, a suitable discrete datatype is needed. For simplicity, we use integer arithmetic for this purpose, although it would also be possible to introduce a finite datatype with sufficiently many individuals. The only integer reasoning that has to be done by the calculus is the decision whether two integer literals are equal or different.

5.1. Non-Interference properties

We assume a fixed extended type environment (∇, I) and an arbitrary, but fixed enumeration $\{v_1, \ldots, v_n\}$ of the program variables PVar. The optimised non-interference formula has the form

$$\{\ \alpha\ \} \Downarrow_C (\nabla, I) \equiv_{\mathrm{def}} \forall k.\ \exists \mathbf{r}.\ \{\, \mathrm{CUpd}(k, \mathbf{r})\, \}[\, \alpha\,]\, \mathrm{CPost}(k, \mathbf{r}) \tag{4}$$

in which k is an integer variable and $\mathbf{r}$ denotes a list $r_1, \ldots, r_n$ of distinct logical variables. As before, $\mathbf{r}$ represents the post-values of the program variables $v_1, \ldots, v_n$. The value of k determines the variable v_k whose independence is supposed to be verified.

$$\frac{\Gamma, \phi \vdash^{\mathrm{dl}} \Delta}{\Gamma \vdash^{\mathrm{dl}} \Delta}\ \textsc{arith}^{\mathrm{dl}} \qquad (\phi \text{ is a theorem of first-order arithmetic})$$

$$\frac{\begin{array}{c}(\Gamma, \phi \vdash^{\mathrm{dl}} \Delta)[x/s] \\ (\Gamma \vdash^{\mathrm{dl}} \phi, \Delta)[x/t]\end{array}}{(\Gamma \vdash^{\mathrm{dl}} \Delta)[x/(\mathit{if}\ \phi\ \mathit{then}\ s\ \mathit{else}\ t)]}\ \textsc{split-if}^{\mathrm{dl}} \qquad \begin{array}{l}(x \notin \mathrm{vars}(\phi), x \text{ not in the} \\ \text{scope of quantifiers,} \\ \text{modalities or updates)}\end{array}$$

$$\begin{array}{l}(\mathit{if}\ \phi\ \mathit{then}\ s_1\ \mathit{else}\ t_1) * (\mathit{if}\ \phi\ \mathit{then}\ s_2\ \mathit{else}\ t_2) \\ \quad \rightarrow^{\mathrm{dl}}\ \mathit{if}\ \phi\ \mathit{then}\ (s_1 * s_2)\ \mathit{else}\ (t_1 * t_2)\end{array} \qquad (* \in \{+, -, \ldots\})$$

Fig. 5. Additional DL rules to handle optimised non-interference formulae.

The update $\mathrm{CUpd}(k, \mathbf{r})$ combines the updates for the different cases in the formula $\{\ \alpha\ \} \Downarrow (\nabla, I)$ (after eliminating the quantifiers):

$$\begin{array}{rcl}\mathrm{CUpd}(k, \mathbf{r}) & =_{\mathrm{def}} & v_1 := V_1(k, \mathbf{r}),\ \ldots,\ v_n := V_n(k, \mathbf{r}) \\ V_i(k, \mathbf{r}) & =_{\mathrm{def}} & \text{if } k = 1 \text{ then } V_{i,1}(r_1) \\ & & \text{else if } k = 2 \text{ then } V_{i,2}(r_2) \\ & & \quad \vdots \\ & & \mathit{else}\ V_{i,n}(r_n) \\ V_{i,j}(r_j) & =_{\mathrm{def}} & \begin{cases} c_{i,j} & \text{for } v_j \in I \text{ or } v_i \in \nabla(v_j) \\ f_{i,j}(r_j) & \text{otherwise} \end{cases}\end{array}$$

where $c_{i,j}$ and $f_{i,j}$ (for $1 \le i, j \le n$) are fresh constant and function symbols. Likewise, the formula $\mathrm{CPost}(k, \mathbf{r})$ performs a local case analysis for checking the different post-conditions of $\{\ \alpha\ \} \Downarrow (\nabla, I)$:

$$\mathrm{CPost}(k, \mathbf{r}) \equiv_{\mathrm{def}} \bigwedge_{1 \le j \le n} \begin{cases} k = j \rightarrow v_j = c_{j,j} & \text{for } v_j \in I \\ k = j \rightarrow v_j = r_j & \text{otherwise.} \end{cases}$$

For each $k \in \{1, \ldots, n\}$, the formula $\exists \mathbf{r}.\ \{\mathrm{CUpd}(k, \mathbf{r})\}[\alpha]\,\mathrm{CPost}(k, \mathbf{r})$ represents one of the conjuncts of $\{\ \alpha\ \} \Downarrow (\nabla, I)$. More formally, we can observe that the optimised non-interference formulae are equivalent to the original formulae, which means that $\{\ \alpha\ \} \Downarrow_C (\nabla, I)$ can be used in the rules $\textsc{If}^{\mathrm{dl}}$ and $\textsc{While}^{\mathrm{dl}}$ as we please:

Lemma 7.

$\{\ \alpha\ \} \Downarrow_C (\nabla, I)$ *is valid if and only if* $\{\ \alpha\ \} \Downarrow (\nabla, I)$ *is valid.*

Instead of the list $\mathbf{r}$ of variables, we could, in principle, also use a single variable r in (4). This would require to apply $\textsc{ex-right}^{\mathrm{dl}}$ multiple times to the same formula in a proof, however, and ultimately lead to proofs of exponential size. As a further optimisation, it is possible to combine the different cases for showing the invariance of program variables I into a single case.

5.2. Proofs with the optimised encoding

Proving programs secure using non-interference formulae (4) requires somewhat more sophisticated first-order reasoning than before. Fig. 5 shows three further rules that are necessary:

- We assume that an oracle – a background reasoner – is available that handles arithmetic for us (rule $\textsc{arith}^{\mathrm{dl}}$).
- The rule $\textsc{split-if}^{\mathrm{dl}}$ handles conditional terms by proof splitting. We only need to apply $\textsc{split-if}^{\mathrm{dl}}$ when the program has disappeared in a proof goal and the post-condition is verified.
- The last rule is an example for a rule that normalises conditional expressions, which can help to keep updates small during a proof. Similar rules are necessary for function application or other operators. Normalisation can also be required before applying the rule $\textsc{abstract}^{\mathrm{dl}}$, because the result of $\textsc{abstract}^{\mathrm{dl}}$ might otherwise be too imprecise (e.g., when applying the rule to a whole expression $V_i(k, \mathbf{r})$).

With the additional rules from Fig. 5, the DL calculus is also complete on non-interference formulae $\{\ \alpha\ \} \Downarrow_C (\nabla, I)$ (in the sense of Theorem 5). This can be seen when examining the proof of Lemma 6, which essentially works by constructing the same proof tree for each conjunct of the old non-interference formulae $\{\ \alpha\ \} \Downarrow (\nabla, I)$. The proofs only differ in the updates in front of programs. A formula $\{\ \alpha\ \} \Downarrow_C (\nabla, I)$ can be proven in the same way, only that more complex first-order reasoning is necessary in the leaves of the proof for handling the case distinctions in $\mathrm{CUpd}(k, r)$ and $\mathrm{CPost}(k, r)$:

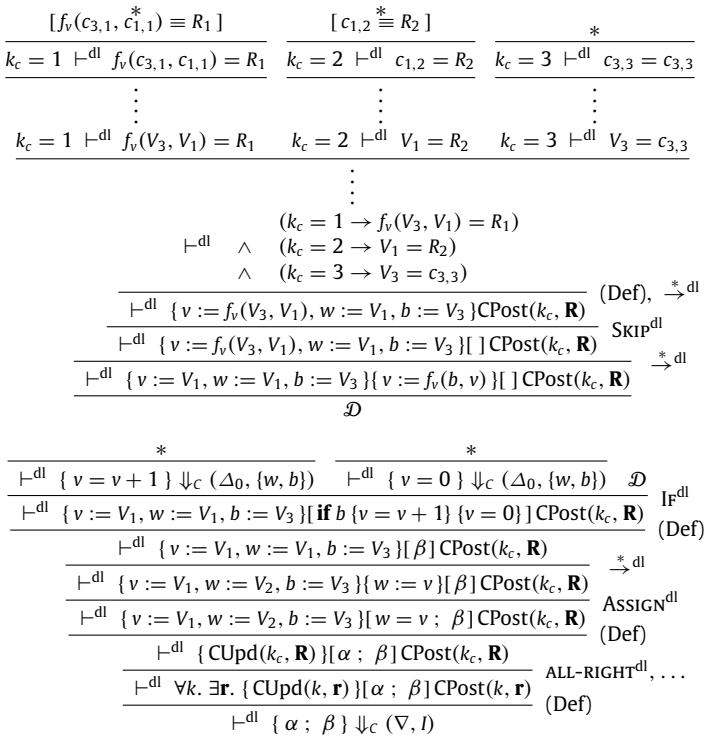

Fig. 6. Example proof using the optimised non-interference formulae. We write α for the program $w = v$ and β for the program **if** b $\{v = v + 1\}$ $\{v = 0\}$. The right-hand sides of the update $\mathrm{CUpd}(k_c, \mathbf{R})$ are abbreviated with V_1, V_2, V_3.

Theorem 8.

$$\bot \vdash^{\mathrm{HS}} \Delta_0\ \{\ \alpha\ \}\ \nabla \quad \textit{implies} \quad \vdash^{\mathrm{dl}}\ \{\ \alpha\ \} \Downarrow_C (\nabla, \emptyset).$$

Example 9. We illustrate the optimized non-interference formulae by verifying that the program "$w = v$; **if** b $\{v = v + 1\}$ $\{v = 0\}$" adheres to the post-typing (∇, I) with $\nabla(v) = \{v, b\}$, $\nabla(w) = \{v\}$, $\nabla(b) = \{b\}$, $I = \{b\}$. For the enumeration $\{v, w, b\}$ of the program variables, the update and the post-condition of the proof obligation are:

$$\mathrm{CUpd}(k, \mathbf{r}) \quad =_{\mathrm{def}} \quad \left\{ \begin{array}{lll} v & := & \text{if } k = 1 \text{ then } c_{1,1} \\ & & \text{else if } k = 2 \text{ then } c_{1,2} \\ & & \text{else } c_{1,3}, \\ w & := & \text{if } k = 1 \text{ then } f_{2,1}(r_1) \\ & & \text{else if } k = 2 \text{ then } f_{2,2}(r_2) \\ & & \text{else } c_{2,3}, \\ b & := & \text{if } k = 1 \text{ then } c_{3,1} \\ & & \text{else if } k = 2 \text{ then } f_{3,2}(r_2) \\ & & \text{else } c_{3,3} \end{array} \right\}$$

$$\begin{array}{lll} \mathrm{CPost}(k, \mathbf{r}) & =_{\mathrm{def}} & (k = 1 \to v = r_1) \\ & & \wedge\ (k = 2 \to w = r_2) \\ & & \wedge\ (k = 3 \to b = c_{3,3}). \end{array}$$

The proof is outlined in Fig. 6 and starts with the non-interference formula $\{\ \alpha\ ;\ \beta\ \} \Downarrow_C (\nabla, I)$ (the two statements of the program are denoted by α and β). As first step, the quantifiers of this formula are eliminated and the assignment $w = v$ is executed, which modifies the update in front of the program. To execute the conditional statement, the typing $(\Delta_0, \{w, b\})$ is used: the variables w and b are not changed in any of the branches, and for both branches the post-value of v only depends on the pre-value of v. We skip the subproofs for the if-branches and only show the branch $\mathcal{D}$, in which the main post-condition is discharged. In $\mathcal{D}$, the effect of the conditional statement is summarised with the update $\mathrm{ifUpd}(b, \Delta_0, \{w, b\}) = (v := f_v(b, v))$ (for a fresh function symbol f_v). After simplifying the update and eliminating the now empty modal operator, the post-condition $\mathrm{CPost}(k_c, \mathbf{R})$ can be expanded and leads to three further branches, one for each program variable. At this point, the conditional terms V_1, V_2, V_3 can be eliminated by repeatedly applying the rules SPLIT-IF$^{\mathrm{dl}}$, ARITH$^{\mathrm{dl}}$ and doing propositional

reasoning (note, that ARITH$^{\text{dl}}$ only has to introduce formulae like $k_c = 1 \to k_c \neq 2$, etc.). Finally, all goals can be closed with the help of CLOSE-EQ$^{\text{dl}}$.

6. Higher precision and delimited information release

Many realistic languages feature exceptions as a means to indicate failure. The occurrence of an exception can also lead to information leakage. Therefore, an information flow analysis for such a language must, at each point where an exception might possibly occur, either ensure that this will indeed not happen at runtime or verify that the induced information flow is benign. The Jif system [6] which implements a security type system for a large subset of the Java language employs a simple data flow analysis to retain a practically acceptable precision w.r.t. exceptions. The data flow analysis can verify the absence of null pointer exceptions and class cast exceptions in certain cases. However, to enhance the precision of this analysis to an acceptable level one is forced to apply a somewhat cumbersome programming style.

The occurrence of exception statements is one example, where we gain something from the fact that our analysis is embedded in a more general program logic: there is no need to stack one analysis on top of another to scale the approach up to bigger languages. Instead, we can uniformly deal with added features, in this case exceptions, within one and the same calculus. In the precise version of the calculus for JavaCard, as implemented in the KeY system [9], exceptions are similar to conditional statements by branching on the condition under which an exception would occur. An uncaught exception is treated as non-termination. As an example, the division v_1/v_2 would have the condition that v_2 is zero:

$$\frac{v_2 \neq 0 \;\vdash^{\text{dl}}\; \{\, w := v_1/v_2 \,\}\, [\,..\;\ldots]\,\phi \qquad v_2 = 0 \;\vdash^{\text{dl}}\; [\,..\;\textbf{throw}\; E\;\ldots]\,\phi}{\vdash^{\text{dl}}\; [\,..\; w = v_1/v_2\;\ldots]\,\phi}$$

("$..\;\ldots$" denotes a context possibly containing exception handlers). If we knew $v_2 \neq 0$ at this point of the proof, implying that the division does in fact not raise an exception, the right branch could be closed immediately. Because our DL calculus stores the values of variables (instead of only their type), as long as no abstraction occurs, the value is usually available:

- rule ASSIGN$^{\text{dl}}$ does not involve abstraction, which means that sequential programs can be executed without loss of information, and
- invariance sets I in non-interference judgments allow us to retain information about unchanged variables across conditional statements and loops.

This can be seen for a program like "$v = 2\;;\;$ **while** $b\,\alpha\;;\; w = w/v$" in which α does not assign to v. By including v in the invariance set for "**while** $b\,\alpha$" we can deduce that $v = 2$ still holds after the loop, and thus be sure that the division will succeed. This is a typical example for a program containing an initialisation part that establishes invariants, and a use part that relies on the invariants. The pattern recurs in many flavours: examples are the initialisation and use of libraries and the well-definedness of references after object creation. We are optimistic to gather further empirical evidence of our claim that increased precision is useful in practice.

6.1. Increasing precision

While our DL calculus is able to maintain state information *across* statements, the rules IF$^{\text{dl}}$ and WHILE$^{\text{dl}}$ lose this information in their first premiss that involves a non-interference proof for the statement *body*. It cannot be deduced, for example, that no exceptions occur in "$v = 2\;;$ **while** $b\,\{w = w/v\}$". As another shortcoming, the branch predicate (the formula that represents the guard of a conditional or loop) is not taken into account, so that absence of exceptions cannot be shown, for example, in the program "**if** $(v \neq 0)\,\{w = 1/v\}$".

One way to remedy this problem is to relax the first premisses in IF$^{\text{dl}}$ and WHILE$^{\text{dl}}$. The idea is to relativize non-interference judgments and introduce *preconditions* ϕ under which the program must satisfy non-interference.

$$\{\,\alpha\,\} \Downarrow (\nabla, I, \phi) \equiv_{\text{def}} \bigwedge_{v \in \text{PVar}} \begin{cases} \dot\forall v_1 \cdots v_n.\; (\phi \to [\,\alpha\,]\, v = u), & v \in I \\ \dot\forall \nabla_v.\; \exists r.\; \dot\forall \nabla_v^C.\; (\phi \to [\,\alpha\,]\, v = r), & v \notin I. \end{cases}$$

In a relativized rule for if-statements, for instance, such a precondition could be used to "pass through" side formulae and state information contained in the update U, as well as to exploit branch predicates: we may use arbitrary preconditions ϕ_1, ϕ_2 in the branches provided that we can show that they hold already before the if-statement is executed:

$$\frac{\begin{array}{cc} \vdash^{\text{dl}}\; \{\,\alpha_1\,\} \Downarrow (\nabla, I, \phi_1) & \vdash^{\text{dl}}\; \{\,\alpha_2\,\} \Downarrow (\nabla, I, \phi_2) \\ \Gamma, \{\,U\,\}\, b = \mathit{TRUE} \;\vdash^{\text{dl}}\; \{\,U\,\}\,\phi_1, \Delta & \Gamma, \{\,U\,\}\, b \neq \mathit{TRUE} \;\vdash^{\text{dl}}\; \{\,U\,\}\,\phi_2, \Delta \\ \multicolumn{2}{c}{\Gamma \;\vdash^{\text{dl}}\; \{\,U\,\}\,\{\,\text{ifUpd}(b, \nabla, I)\,\}\,[\,\ldots]\,\phi, \Delta} \end{array}}{\Gamma \;\vdash^{\text{dl}}\; \{\,U\,\}\,[\,\textbf{if}\; b\;\alpha_1\;\alpha_2\;;\;\ldots]\,\phi, \Delta}\;.$$

An even more interesting usage of relativized non-interference statements is to handle delimited information release in the style of Darvas et al. [4], i.e., situations in which non-interference does not strictly hold and some well-defined information about secret values may be released. The general approach of [4] is to define a partition of the program state

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[f_l'(TRUE) \equiv R]^*}}{odd(f_h(R)) \vdash^{dl} f_l'(TRUE) = R}\ \text{CLOSE-EQ}^{dl}}{odd(f_h(R)) \vdash^{dl} f_l'(odd(f_h(R))) = R}\ \text{APPLY-EQ}^{dl}}{odd(f_h(R)) \vdash^{dl} \{l := f_l(R), h := f_h(R)\}\,\{l := f_l'(odd(h))\}\,l = R}\ \xrightarrow{*}{}^{dl}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdots}}{\vdash^{dl} \{l = 0\} \Downarrow (\nabla, \{h\})}^{*} \quad \dfrac{\dfrac{}{\vdots}}{\vdash^{dl} \{l = 1\} \Downarrow (\nabla, \{h\})}^{*} \quad \mathcal{D}}{odd(f_h(R)) \vdash^{dl} \{l := f_l(R), h := f_h(R)\}\,[\alpha]\,l = R}\ \text{IF}^{dl}}{\vdots \quad \vdash^{dl} \{l := f_l(R), h := f_h(R)\}\,(odd(h) \to [\alpha]\,l = R)}\ \xrightarrow{*}{}^{dl}, \text{IMP-RIGHT}^{dl}$$

$$\dfrac{\dots \qquad \vdash^{dl} \exists r.\, \dot\forall l.\, \dot\forall h.\, (odd(h) \to [\alpha]\,l = r)}{\vdash^{dl} \{\alpha\} \Downarrow (\nabla, \{h\}, odd(h))}\ \text{EX-RIGHT}^{dl}, \text{ALL-RIGHT}^{dl}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[c_l + c_s \equiv R]^*}}{c_s = f_x(R) + f_y(R) \vdash^{dl} c_l + c_s = R}\ \text{CLOSE-EQ}^{dl}}{c_s = f_x(R) + f_y(R) \vdash^{dl} c_l + f_y(R) + f_x(R) = R}\ \text{APPLY-EQ}^{dl}, \text{ARITH}^{dl}}{c_s = f_x(R) + f_y(R) \vdash^{dl} \{l := c_l + f_y(R) + f_x(R),\ U\}\,[\,]\,l = R}\ \text{SKIP}^{dl}}{c_s = f_x(R) + f_y(R) \vdash^{dl} \{l := c_l,\ U\}\,[l = l + y + x]\,l = R}\ \text{ASSIGN}^{dl}}{\vdash^{dl} \{l := c_l,\ U\}\,(s = x + y \to [\alpha]\,l = R)}\ \text{IMP-RIGHT}^{dl}$$

$$\dfrac{\dots \qquad \vdash^{dl} \dot\forall l.\, \dot\forall s.\, \exists r.\, \dot\forall x.\, \dot\forall y.\, (s = x + y \to [\alpha]\,l = r)}{\vdash^{dl} \{\alpha\} \Downarrow (\nabla, \{x, y, s\}, s = x + y)}\ \text{ALL-RIGHT}^{dl}, \dots$$

Fig. 7. Non-interference proof with delimited information release: the precondition $odd(h)$ entails that (only) the parity of h is allowed to leak into l. A similar proof is required for $\neg odd(h)$. For sake of brevity, we use *odd* both as function and predicate, and only in one step (APPLY-EQ^{dl}) make use of the fact that $odd(f_h(R))$ actually represents the equation $odd(f_h(R)) = TRUE$.

Fig. 8. Non-interference proof in which the sum $x + y$ is declassified by specifying that the value of the variable s may flow into l: $\nabla(l) = \{l, s\}$. Applications of the rule $\xrightarrow{*}{}^{dl}$ are not shown explicitly, and U is written as abbreviation for the update $s := c_s, x := f_x(R), y := f_y(R)$.

space and to prove the security of a program separately for each partition. This means that information about the partition membership may freely flow into all variables.

We can use the same method in our DL calculus. As an example, Fig. 7 shows in parts a non-interference proof with delimited information release for the program "$\alpha =$ **if** $(odd(h))\ \{l = 0\}\ \{l = 1\}$". The two partitions that are considered are defined by the formulae $odd(h)$ and $\neg odd(h)$, i.e., the parity of the secret variable h is declassified. The typing ∇ is given by $\nabla(l) = \emptyset$, $\nabla(h) = \{h\}$, indicating that only declassified information flows into l.

6.2. Declassification of expressions

The approach that is sketched in the previous paragraph has the disadvantage that a separate proof is necessary for each partition, multiplying the verification effort. A formulation that is more direct and closer to implementations like in Jif [6] would allow to downgrade the security level of the initial values of certain expressions, which henceforth can be used more liberally in the program than the individual variables that the expressions consist of. Non-interference judgments with preconditions can specify such information release quite naturally: in order to describe that the expressions $e_1, \dots, e_k$ are declassified during the execution of a program α, we assume that k fresh variables $w_1, \dots, w_k \in \text{PVar}$ are available that do not occur in α. The security of α is then simply expressed by:

$$\{\alpha\} \Downarrow (\nabla, I, w_1 = e_1 \wedge \dots \wedge w_k = e_k).$$

In addition to the normal independence specification, the type environment ∇ can now also allow that the value of an expression e_i may flow into a variable v by including w_i in its type: $w_i \in \nabla(v)$.

We illustrate the approach by verifying the program "$\alpha = (l = l + y + x)$" under the assumption that the sum $x + y$ is declassified and may flow into l, while the values of the variables x and y stay confidential. Therefore, we choose the precondition $s = x + y$, a type environment ∇ with $\nabla(l) = \{l, s\}$ and the invariance set $I = \{x, y, s\}$. Parts of the proof are shown in Fig. 8. The most important step in the proof is the reasoning about arithmetic equations, which allows to simplify the post-value $c_l + f_y(R) + f_x(R)$ of l to the expression $c_l + c_s$ that is trivially unifiable with R. As always, the fact that R can be eliminated from the symbolic post-value embodies the independence from confidential information.

7. Conclusion, related and future work

In this paper we made a formal connection between type-based and logic-based approaches to information flow analysis. We showed that every program that is typable in Hunt and Sands' type system [16] has a corresponding proof in an abstract version of dynamic logic. We argued that an integrated logic-based approach fits well into a proof-carrying code framework for establishing security policies of mobile software. In order to support this claim we showed how to increase the precision of the program logic, for example, to express declassification. We also showed that in many cases the program logic formulation is even *value-sensitive*. This allows us to retain concrete values of program variables as long as the usage of the variable does not require its abstraction. This can improve the quality of the analyses (in the sense that fewer secure programs are rejected) considerably.

7.1. Related work

The background for our work is provided by a number of recent type-based and logic-based approaches to information flow analysis [1,18,4,16]. Our concrete starting points were the flow-sensitive type system of Hunt and Sands [16] and the characterisation of non-interference of Darvas et al. [4]. Amtoft and Banerjee [18] devised an analysis with a very logic-like structure that is, however, not more precise than the type system of Hunt and Sands. In an early paper Andrews and Reitman [23] developed a flow logic – one may also consider it a security type system – for proving information flow properties of concurrent Pascal programs. They outline a combination of their flow logic with regular Hoare logic, but keep the formulae for both logics separated. Joshi and Leino [2] give logical characterisations of the semantic notion of information flow, and their presentation in terms of Hoare triples is similar in spirit to our basic formulation. Their results do, however, not provide means to aid automated proofs of these triples. Beringer et al. [24] presented a logic for resource consumption whose proof rules and judgements are derived from a more general program logic; both logics are formalised in the Isabelle/HOL proof assistant. Their approach is similar in spirit to the one presented here, since the preciseness of their derived logic is compared to an extant type system for resource consumption.

More recently, Beringer and Hofmann [25] suggested an alternative approach for a VDM-style logic based on self-composition. The introduction of a predicate operator Sec allows to avoid double execution of the program to be proven secure. Therefore, it is sufficient to show that $\mathrm{Sec}(\phi)$ holds for some formula ϕ. A meta-result allows to conclude the security of the target program utilising $\mathrm{Sec}(\phi)$ twice in its proof. The construction of the formula ϕ and of a suitable judgement in the logic is obtained automatically from a type derivation. This idea is related to the work of Pan [26, Chapter 2], where a suitable first-order security predicate is synthesised from the verification conditions obtained from symbolic execution of the target program. Beringer and Hofmann embed the type-based secure information flow systems of Volpano-Smith and of Hunt and Sands using different "instantiations" of Sec. They have an extension of their program logic for heap structures, albeit only for a flow-insensitive security type system. On the other hand, they do not cover delimited information release as outlined in Section 6. The size of the constructed formula ϕ relative to the length of the type derivation is open.

This work is concerned with connecting type-based and deduction-based systems for non-interference in a flow-sensitive setting. There is a large body of work on non-interference outside of this particular setting, for example, process algebras such as CCS (e.g., [27]), CSP (see [28] for an overview), π-calculus, the spi calculus, and other event-based systems (e.g., [29]). Of these, the closest approach to ours is that of Bossi et al. [30,31] who model non-interference with unwinding conditions over a security process algebra language. These can then be decided by general methods for universal first-order formulas over the reals. In [31] a concurrent language and different notion of non-interference from the present paper is used. It is open whether general decision procedures for fragments of first-order logic are efficient enough in practice.

The literature on flow-insensitive systems (such as Jif and FlowCaml) is vast, therefore, we refer to the survey of Myers and Sabelfeld [1].

None of the cited work combines abstract and precise reasoning as we propose in Section 6, although it would probably be possible in the framework of Beringer and Hofmann.

7.2. Future work

We have not formally investigated the precise complexity of the translation of HS type derivations to DL proofs (Theorems 5 and 8) and the size of resulting proofs. It appears that the usage of the optimised proof obligations $\{\ \alpha\ \} \Downarrow_C (\nabla, I)$ (Section 5) leads to proofs that grow at most quadratically in the number of program variables, and that grow linearly in the program size for a fixed number of program variables.

The present work is a basic framework for the integration of type-based and logic-based information-flow analysis. It demonstrates that a uniform logical treatment of type-based and logic-based analysis is feasible and advantageous. In addition to non-interference and declassification, more complex security policies need to be looked at. It has to be seen how well the notion of abstraction presented in this paper is suited to express these. We also want to extend the program logic to cover at least JavaCard, based on the axiomatisation in [9] and the implementation in the program verifier KeY [32]. Ideas towards this goal have been worked out in [26], parts of which are also presented in [5]. Finally, a suitable notion of proof certificate and proof checking for proof-carrying code must be derived for dynamic logic proofs of security policies. This is a major undertaking and beyond the scope of this paper.

Acknowledgments

We would like to thank Dave Sands for inspiring discussions. Andrei Sabelfeld reminded us of declassification and helped with the literature on information flow. Thanks to Tarmo Uustalu for pointing out [23]. Richard Bubel provided many valuable remarks. The comments of the anonymous reviewers helped to improve the paper in several respects.

Appendix

The following sections contain shortened proofs to most lemmas and theorems of the paper. Complete versions of the proofs can be found in [5].

A.1. Proof of Theorem 1 (Soundness)

Substitution – as used in rule ABSTRACT$^{\text{dl}}$ – must be handled with care in dynamic logic, due to the presence of modal operators. Therefore, one only gets a restricted version of a substitution theorem:

Lemma 10 (*Substitution in Dynamic Logic*). *Let $S = (D, I)$ be a structure and t_1, t_2 terms. Suppose that for all program variable assignments δ, δ', and all variable assignments β, it is the case that $val_{S,\beta,\delta}(t_1) = val_{S,\beta,\delta'}(t_2)$. Then for all formulae ϕ of dynamic logic and all (program) variable assignments β, δ one has $val_{S,\beta,\delta}(\phi[x/t_1]) = val_{S,\beta,\delta}(\phi[x/t_2])$.*

Proof (*Theorem* 1). The proofs for the rules relating to predicate logic are standard and therefore omitted. For a description of update application rules and soundness proofs see [21]. The interesting cases are ABSTRACT$^{\text{dl}}$, WHILE$^{\text{dl}}$ and IF$^{\text{dl}}$, of which we present the first two.

ABSTRACT$^{\text{dl}}$. We apply Lemma 10. Therefore, given a structure $S = (D, I)$ and program variable assignment δ invalidating the conclusion of ABSTRACT$^{\text{dl}}$ we construct a structure $S_f = (D, I_f)$ such that (i) I_f coincides with I apart from the interpretation $I_f(f)$, and (ii) $val_{S_f,\beta,\delta}(t) = val_{S_f,\beta,\delta}(f(\text{vars}(t)))$ for all variable assignments β, δ. Obviously, S_f is uniquely defined by these two conditions. By Lemma 10 and the fact that f is fresh we then obtain the following identities, implying that S_f, δ invalidate the premiss of ABSTRACT$^{\text{dl}}$ for all β:

$$\begin{aligned} val_{S,\beta,\delta}((\Gamma \vdash^{\text{dl}} \Delta)[x/t]) &= val_{S_f,\beta,\delta}((\Gamma \vdash^{\text{dl}} \Delta)[x/t]) \\ &= val_{S_f,\beta,\delta}((\Gamma \vdash^{\text{dl}} \Delta)[x/f(\text{vars}(t))]). \end{aligned}$$

WHILE$^{\text{dl}}$. We ignore any possible update $\{U\}$ that might occur in front of the formulae in the second premiss and the conclusion as this does not add any interesting detail. We assume the first premisses of WHILE$^{\text{dl}}$ and that the conclusion is invalidated by some δ for all β: $val_{S,\delta,\beta}([\textbf{while}\ b\ \alpha]\phi) = ff$, so that $[\![\textbf{while}\ b\ \alpha]\!]^S\delta = \delta' (\neq \perp)$ and $val_{S,\delta',\beta}(\phi) = ff$. We need to show that there exists S' agreeing with S apart from the interpretation of the fresh f_v s. t. $val_{S',\delta,\beta}(f_v(\gamma^*_\nabla(\nabla))) = ([\![\textbf{while}\ b\ \alpha]\!]^S\delta)(v)$, which would invalidate the second premiss of the rule. From the first set of premisses we obtain, for all v and all δ, δ' that agree on all $u \in \gamma^*_\nabla(\nabla)(v)$, that $([\![\textbf{if}\ b\ \alpha\ ]\!]^S\delta)(v) = ([\![\textbf{if}\ b\ \alpha\ ]\!]^S\delta')(v)$.

Importantly, $\gamma^*_\nabla(\nabla)$ has a closure property that is ensured by the side condition $v \in \nabla(v)$ for all v. Namely, $w \in \gamma^*_\nabla(\nabla)(v)$ implies $\gamma^*_\nabla(\nabla)(w) \subseteq \gamma^*_\nabla(\nabla)(v)$: if a variable depends on another, the latter's dependencies are included in the former's. This yields the equality for *all* dependencies of v:

$$([\![\textbf{if}\ b\ \alpha\ ]\!]^S\delta)(u) = ([\![\textbf{if}\ b\ \alpha\ ]\!]^S\delta')(u), \quad \text{f.a. } u \in \gamma^*_\nabla(\nabla)(v). \tag{A.1}$$

The interpretations of the f_v are definable as least fixed-points of an ascending chain of functions.[5] We show the construction of f_v for a given variable v. Therefore, it is convenient to work with a semantic function for loops that is restricted to the value of a single variable.

$$w^0_v(\delta) = \perp, \qquad w^{n+1}_v(\delta) = \begin{cases} (w^n_v)_\perp([\![\alpha]\!]^S\delta) & \text{for } val_{S,\delta}(b) = val_S(TRUE) \\ \delta(v) & \text{otherwise.} \end{cases}$$

Now let $|\nabla(v)| = k$ and inductively assume there is a function $f^n_v : D^k \to D_\perp$ s. t. $w^n_v(\delta) \sqsubseteq f^n_v(d_1, \ldots, d_k)$ f. a. δ with $\delta(\nabla_v[j]) = d_j$, $1 \le j \le k$ (in particular, this states that w^n_v yields the same results (or $\perp$) for all such δ); then we construct appropriate $f^{n+1}_v \sqsupseteq f^n_v$. The essential point to show is that $w^{n+1}_v(\delta) = w^{n+1}_v(\delta')$ for all δ, δ' that agree on $\nabla(v)$ and where $w^{n+1}_v(\delta) \neq \perp \neq w^{n+1}_v(\delta')$. Then we know that for all $d_1, \ldots, d_k$ and all assignments δ with $\delta(\nabla_v[j]) = d_j$ there is a value r such that $w^{n+1}_v(\delta) = r$ or $w^{n+1}_v(\delta) = \perp$, meaning there is at most one final value of v if one fixes the initial values of the ∇_v to $d_1, \ldots, d_k$. We let $f^{n+1}_v(d_1, \ldots, d_k)$ yield that value, or $\perp$ if there is no such value.

Let δ, δ' agree on $\nabla(v)$ and, crucially, thereby also on $\gamma^*_\nabla(\nabla)(v)$, since the latter set is a subset of the former by virtue of the side condition on WHILE$^{\text{dl}}$. To show $w^{n+1}_v(\delta) = w^{n+1}_v(\delta')$ we consider the three possible cases:

[5] The so obtained function $f_v : D^{|\nabla(v)|} \to D_\perp$ can easily be converted to a function of the right type $D^{|\nabla(v)|} \to D$ by remapping all elements on which f_v yields bottom to some arbitrary value in D, since we consider a terminating execution.

- $val_{S,\delta}(b) = val_{S,\delta'}(b) \neq val_S(TRUE)$: since $v \in \gamma^*_\nabla(\nabla)(v)$, we have $w_v^{n+1}(\delta) = \delta(v) = \delta'(v) = w_v^{n+1}(\delta')$.
- $val_{S,\delta}(b) = val_{S,\delta'}(b) = val_S(TRUE)$: we obtain $w_v^{n+1}(\delta) = [\![\alpha]\!]^S\delta = \delta_1$ and $w_v^{n+1}(\delta) = [\![\alpha]\!]^S\delta' = \delta'_1$ where, by (A.1), δ_1, δ'_1 again agree on $\gamma^*_\nabla(\nabla)$, hence $\delta_1(v) = \delta'_1(v)$. The slightly more involved case.
- has $val_{S,\delta}(b) \neq val_S(TRUE) = val_{S,\delta'}(b)$, so that for δ the terminating case is chosen ($w_v^{n+1}(\delta) = \delta(v)$), and for δ' the evaluation continues recursively. The assumption $w_v^{n+1}(\delta') \neq \bot$ ensures there is an $m \leq n$ s. t. $([\![\alpha]\!]^S)^m\, \delta'\, (v) = w_v^n([\![\alpha]\!]^S\delta')$, i.e. to obtain the result of $w_v^{n+1}(\delta')$ we 'run α on δ' m times'. But by (A.1) we know that running α on an assignment that agrees with δ on $\gamma^*_\nabla(\nabla)$ (as δ' does) yields an assignment that again agrees with δ on these variables. By an easy induction we finally see that $([\![\alpha]\!]^S)^m\, \delta'$ agrees with δ on the desired domain, too. □

A.2. Auxiliary results about the type system of Hunt and Sands

In order to show Lemma 2 (that allows to eliminate context types) and Lemma 4 (equivalence of the systems HS and cf), we first need a number of further results about Hunt and Sands' type system.

Lemma 11. *It is possible to increase the type of variables in a typing judgement by joining its type with the context p: we write $\nabla_{x\uparrow p}$ for the typing $\nabla[x \mapsto p \sqcup \nabla(x)]$. Then the following holds:*

$$p' \vdash^{\text{HS}} \nabla\ \{\ \alpha\ \}\ \nabla' \quad \textit{and} \quad p \sqsubseteq p' \quad \textit{implies} \quad p' \vdash^{\text{HS}} \nabla_{x\uparrow p}\ \{\ \alpha\ \}\ \nabla'_{x\uparrow p}.$$

Proof. By induction on the structure of type derivations. We can observe that by simply lifting *all* typings ∇ to $\nabla_{x\uparrow p}$ in a given derivation for a judgement $p' \vdash^{\text{HS}} \nabla'\ \{\ \alpha\ \}\ \nabla''$, we obtain a derivation for $p' \vdash^{\text{HS}} \nabla'_{x\uparrow p}\ \{\ \alpha\ \}\ \nabla''_{x\uparrow p}$. □

Lemma 12. *Given a valid typing judgement and type p, one retains a valid judgement when lifting the context and the post-type of assigned variables by p:*

$$t \vdash^{\text{HS}} \nabla\ \{\ \alpha\ \}\ \nabla' \quad \textit{implies} \quad t \sqcup p \vdash^{\text{HS}} \nabla\ \{\ \alpha\ \}\ \nabla'_{\alpha\uparrow p}$$

where

$$\nabla'_{\alpha\uparrow p}(x) \;=\; \begin{cases} \nabla'(x) \sqcup p & \textit{for } x \in \mathit{Lhs}(\alpha) \\ \nabla'(x) & \textit{otherwise}. \end{cases}$$

Proof. By induction on the structure of derivations. All cases except for $\text{Seq}^{\text{cfree}}$ and $\text{While}^{\text{cfree}}$ are immediate, and the latter ones basically follow from Lemma 11. For the $\text{While}^{\text{cfree}}$ case, we are given a derivation of the judgement $t \vdash^{\text{HS}} \nabla\ \{\ \textbf{while}\ E\ \alpha\ \}\ \nabla$, where $\nabla \vdash E : t'$, and we need to show $t \sqcup p \vdash^{\text{HS}} \nabla\ \{\ \textbf{while}\ E\ \alpha\ \}\ \nabla_{\alpha\uparrow p}$. By the induction hypothesis we know $t' \sqcup t \sqcup p \vdash^{\text{HS}} \nabla\ \{\ \alpha\ \}\ \nabla_{\alpha\uparrow p}$ which we can extend to a derivation of $t' \sqcup t \sqcup p \vdash^{\text{HS}} \nabla_{\alpha\uparrow p}\ \{\ \alpha\ \}\ \nabla_{\alpha\uparrow p}$ by Lemma 11. Two rule applications of While^{HS} and Sub^{HS} respectively yield the required derivation. □

Lemma 13. *Variables that are not assigned in a program are not declassified:*

$$p \vdash^{\text{HS}} \nabla\ \{\ \alpha\ \}\ \nabla' \quad \textit{and} \quad v \notin \mathit{Lhs}(\alpha) \quad \textit{implies} \quad \nabla(v) \sqsubseteq \nabla'(v).$$

Proof. (Lemma 2) "$\Longrightarrow$" The first conjunct on the right-hand side can obviously be obtained from the derivation on the left by a single application of Sub^{HS}. To conclude, one shows the following implication by induction on the set of valid typing judgments, referring to Lemma 13:

$$p \vdash^{\text{HS}} \nabla\ \{\ \alpha\ \}\ \nabla' \quad \Longrightarrow \quad \text{f.a. } v \in \mathit{Lhs}(\alpha).\ p \sqsubseteq \nabla'(v).$$

"$\Longleftarrow$" Follows directly from Lemma 12 (with $t = \bot$), because given that for all $v \in \mathit{Lhs}(\alpha).\ p \sqsubseteq \nabla'(v)$ one has $\nabla'_{\alpha\uparrow p} = \nabla'$, so that the two statements coincide.

A.3. Proof of Lemma 4 (Equivalence of the Systems HS and cf)

We show Lemma 4 through a number of transformation steps, starting with system HS and eventually ending with system cf. Altogether, the proof of Lemma 4 is split into three parts:

$$\begin{aligned} \bot \vdash^{\text{HS}} \Delta_0\ \{\ \alpha\ \}\ \nabla \quad &\text{iff} \quad \vdash^{\text{cfree}} \Delta_0\ \{\ \alpha\ \}\ \nabla \\ &\text{iff} \quad \vdash^{\text{cfa}} \Delta_0\ \{\ \alpha\ \}\ \nabla \\ &\text{iff} \quad \vdash^{\text{cf}} \Delta_0\ \{\ \alpha\ \}\ \nabla. \end{aligned}$$

System cfree (Fig. A.1) is a slight modification of Hunt and Sands' original system where we have removed the context p from typings and replaced it by side conditions relating to the type of assigned variables. This step is crucial since the side condition is very natural to express in the DL calculus, which is not the case for the context. The equivalence of the systems HS and cfree can be shown by induction on the set of valid type judgements and using Lemma 2.

$$\frac{}{\vdash^{\text{cfree}} \nabla\ \{\ \}\ \nabla}\ \textsc{Skip}^{\text{cfree}}$$

$$\frac{\nabla \vdash E : t}{\vdash^{\text{cfree}} \nabla\ \{\ v = E\ \}\ \nabla[v \mapsto t]}\ \textsc{Assign}^{\text{cfree}}$$

$$\frac{\vdash^{\text{cfree}} \nabla\ \{\ \alpha_1\ \}\ \nabla' \qquad \vdash^{\text{cfree}} \nabla'\ \{\ \alpha_2\ \}\ \nabla''}{\vdash^{\text{cfree}} \nabla\ \{\ \alpha_1\ ;\ \alpha_2\ \}\ \nabla''}\ \textsc{Seq}^{\text{cfree}}$$

$$\frac{\nabla \vdash b : t \qquad \vdash^{\text{cfree}} \nabla\ \{\ \alpha_i\ \}\ \nabla' \quad (i = 1, 2)}{\vdash^{\text{cfree}} \nabla\ \{\ \textbf{if}\ b\ \alpha_1\ \alpha_2\ \}\ \nabla'}\ \textsc{If}^{\text{cfree}} \quad \begin{array}{l}\text{f.a. } v \in \mathit{Lhs}(\alpha_1).\ t \sqsubseteq \nabla'(v)\\ \text{f.a. } v \in \mathit{Lhs}(\alpha_2).\ t \sqsubseteq \nabla'(v)\end{array}$$

$$\frac{\nabla \vdash b : t \qquad \vdash^{\text{cfree}} \nabla\ \{\ \alpha\ \}\ \nabla}{\vdash^{\text{cfree}} \nabla\ \{\ \textbf{while}\ b\ \alpha\ \}\ \nabla}\ \textsc{While}^{\text{cfree}} \quad \text{f.a. } v \in \mathit{Lhs}(\alpha).\ t \sqsubseteq \nabla(v)$$

$$\frac{\vdash^{\text{cfree}} \nabla_1\ \{\ \alpha\ \}\ \nabla_1'}{\vdash^{\text{cfree}} \nabla_2\ \{\ \alpha\ \}\ \nabla_2'}\ \textsc{Sub}^{\text{cfree}} \quad \nabla_2 \sqsubseteq \nabla_1,\ \nabla_1' \sqsubseteq \nabla_2'$$

Fig. A.1. Intermediate flow-sensitive type system cfree.

$$\frac{}{\vdash^{\text{cfa}} \nabla\ \{\ \}\ \nabla'}\ \textsc{Skip}^{\text{cfa}} \quad \nabla \sqsubseteq \nabla'$$

$$\frac{\nabla \vdash E : t \qquad \vdash^{\text{cfa}} \nabla[v \mapsto t]\ \{\ \ldots\ \}\ \nabla'}{\vdash^{\text{cfa}} \nabla\ \{\ v = E\ ;\ \ldots\ \}\ \nabla'}\ \textsc{Assign}^{\text{cfa}}$$

$$\frac{\nabla \vdash b : t \qquad \vdash^{\text{cfa}} \nabla'\ \{\ \ldots\ \}\ \nabla'' \quad \vdash^{\text{cfa}} \nabla\ \{\ \alpha_i\ \}\ \nabla' \quad (i = 1, 2)}{\vdash^{\text{cfa}} \nabla\ \{\ \textbf{if}\ b\ \alpha_1\ \alpha_2\ ;\ \ldots\ \}\ \nabla''}\ \textsc{If}^{\text{cfa}} \quad \begin{array}{l}\text{f.a. } v \in \mathit{Lhs}(\alpha_1).\ t \sqsubseteq \nabla'(v)\\ \text{f.a. } v \in \mathit{Lhs}(\alpha_2).\ t \sqsubseteq \nabla'(v)\end{array}$$

$$\frac{\nabla' \vdash b : t \qquad \vdash^{\text{cfa}} \nabla'\ \{\ \ldots\ \}\ \nabla'' \quad \vdash^{\text{cfa}} \nabla'\ \{\ \alpha\ \}\ \nabla'}{\vdash^{\text{cfa}} \nabla\ \{\ \textbf{while}\ b\ \alpha\ ;\ \ldots\ \}\ \nabla''}\ \textsc{While}^{\text{cfa}} \quad \begin{array}{l}\nabla \sqsubseteq \nabla'\\ \text{f.a. } v \in \mathit{Lhs}(\alpha).\ t \sqsubseteq \nabla'(v)\end{array}$$

Fig. A.2. Intermediate flow-sensitive type system cfa.

System cfa (Fig. A.2) is obtained by further modifying the system cfree to make it more similar to the DL calculus, which always operates on the first statement of a program (the *active statement*, cf. [9]). The $\textsc{Seq}^{\text{cfree}}$ and $\textsc{Sub}^{\text{cfree}}$ rules are integrated into the other rules. It is easy to show that each cfa rule can be simulated in terms of cfree rules, which proves the soundness of cfa. On the other hand, an arbitrary cfree derivation can be normalised, employing the associativity of sequential composition, and then translated into a cfa derivation.

Finally, the type system is brought into a shape that directly corresponds to our DL calculus (system cf in Fig. 4). The difference to cfa is that we only work with typings of the form $\vdash^{\text{cf}} \Delta_0\ \{\ \cdot\ \}\ \nabla'$. We can show the equivalence of cfa and cf primarily using Lemma 3 for the type system HS (Lemma 6.8 about canonical derivations in [16]), which also holds for the equivalent system cfa:

$$\vdash^{\text{cfa}} \nabla\ \{\ \alpha\ \}\ \nabla' \quad \text{iff} \quad \vdash^{\text{cfa}} \Delta_0\ \{\ \alpha\ \}\ \gamma^*_\nabla(\nabla').$$

A.4. Proof of Lemma 6

For showing that derivations in cf can be translated to proofs in the DL calculus, we first need a bit of further notation for updates. For an update U and a term s, we write $U\,[s]$ for the (unique) irreducible/update-free term s' that is obtained by repeatedly applying rules of Fig. 3: $\{\ U\ \}\ s \xrightarrow{*}{}^{\text{dl}} s'$.

Further, for an update U, a type $t \subseteq \text{PVar}$ and a logical variable $R \in \text{LVar}$, we write $\mathit{mrk}(t, R, U)$ if the following identity holds:

$$\{v \in \text{PVar} \mid R \in \text{vars}(U\,[v])\} = \text{PVar}\backslash t. \tag{A.2}$$

Intuitively, this means that all variables $w \in \text{PVar}\backslash t$ whose interference is prohibited are "marked" by U with a free variable R. Removing the quantifiers in a non-interference statement like

$$\forall u_1\, u_2\ \exists r.\ \forall u_3\, u_4.\ \{\ v_i := u_i\ \}_{1 \le i \le 4}\ [\,p\,]\ (v_1 = r)$$

using rules $\textsc{all-right}^{\text{dl}}$ and $\textsc{ex-right}^{\text{dl}}$ exactly creates this situation (in the example for $t = \nabla(v_1) = \{v_1, v_2\}$).

Referring to the last rule of Fig. 3, we will denote the update obtained by *sequentially composing* two updates U_1 and $U_2 = v_1 := t_1, \ldots, v_k := t_k$ by

$$U_1; U_2 =_{\text{def}} U_1,\ v_1 := \{\ U_1\ \}\ t_1, \ldots, v_k := \{\ U_1\ \}\ t_k.$$

Recall the concretisation operator γ_∇ (cf. [16]) used in the type system cf:

$$\gamma_\nabla(t) =_{\text{def}} \{v \mid \nabla(v) \sqsubseteq t\}, \qquad \gamma^*_{\nabla_1}(\nabla_2)(v) =_{\text{def}} \gamma_{\nabla_1}(\nabla_2(v)).$$

There is an immediate relationship between sequential update composition and γ_∇, which is the key property that enables to simulate type derivations in the DL calculus:

Lemma 14. *Suppose that for an update U' and a typing $\nabla' : \text{PVar} \to \mathcal{P}(\text{PVar})$ the following property holds:*

$$\textit{f.a. } v \in \text{PVar}. \qquad \nabla'(v) = \text{vars}(U'[v]) \cap \text{PVar} \quad \textit{and} \quad R \notin \text{vars}(U'[v]).$$

Then

$$mrk(t, R, U) \quad \textit{implies} \quad mrk(\gamma_{\nabla'}(t), R, (U; U')).$$

Proof. The stated implication follows almost immediately from (A.2) and the following equivalence (for arbitrary updates U, U' and variables $v \in \text{PVar}$, $R \in \text{LVar}$), which can be proven by induction on $U'[v]$:

$$R \in \text{vars}((U; U')[v]) \quad \text{iff} \quad \text{there is } x \in \text{vars}(U'[v]) \text{ with } R \in \text{vars}(U[x]).$$

Proof (*Lemma* 6). We show the stronger implication

$$\vdash^{\text{cf}} \Delta_0 \{\alpha\} \nabla \quad \Longrightarrow \quad I \cap Lhs(\alpha) = \emptyset \quad \Longrightarrow \quad \vdash^{\text{dl}} \{\alpha\} \Downarrow (\nabla, I)$$

by noetherian induction on the program α, using the sub-program-order: For showing the implication for a program α, we will assume that it holds for all programs $\alpha' \neq \alpha$ that literally occur as part of α.

We then first decompose α into a list $\alpha = \alpha_1 ; \ldots ; \alpha_m$ of statements ($m = 0$ is possible) and assume that $\vdash^{\text{cf}} \Delta_0 \{\alpha\} \nabla$ and $I \cap Lhs(\alpha) = \emptyset$. The formula $\{\alpha\} \Downarrow (\nabla, I)$ leads to two kinds of proof obligations:

Non-interference obligations: For $PO = \dot\forall \nabla_v.\ \dot\exists r.\ \dot\forall \nabla_v^C.\ [\alpha] r = v$ and $v \notin I$, we prove by induction on a $k \in \mathbb{N}$, $k \leq m$ the following properties:

- There is a dl proof tree with PO as root that has exactly one open branch $\vdash^{\text{dl}} \{U\} [\alpha_{k+1} ; \ldots ; \alpha_m] R = v$, where U is an update.
- For some typing ∇' with $mrk(\nabla'(v), R, U)$, there is a type derivation of $\vdash^{\text{cf}} \Delta_0 \{\alpha_{k+1} ; \ldots ; \alpha_m\} \nabla'$ that corresponds to the open goal.

Invariance obligations: For $PO = \dot\forall v_1 \cdots v_n.\ \forall u.\ \{v := u\}[\alpha] u = v$ and $v \in I$, we prove by induction on a $k \in \mathbb{N}$, $k \leq m$ the following properties:

- There is a dl proof tree with PO as root that has exactly one open branch $\vdash^{\text{dl}} \{U\} [\alpha_{k+1} ; \ldots ; \alpha_m] u_c = v$, where U is an update with $U[v] = u_c$.
- There is a type derivation of $\vdash^{\text{cf}} \Delta_0 \{\alpha_{k+1} ; \ldots ; \alpha_m\} \nabla'$ that corresponds to the open goal.

References

[1] A. Sabelfeld, A.C. Myers, Language-based information-flow security, IEEE Journal on Selected Areas in Communications 21 (1) (2003) 5–19.
[2] R. Joshi, K.R.M. Leino, A semantic approach to secure information flow, Science of Computer Programming 37 (1–3) (2000) 113–138.
[3] G. Barthe, P.R. D'Argenio, T. Rezk, Secure Information Flow by Self-Composition, in: R. Foccardi (Ed.), Proceedings of CSFW'04, IEEE Press, Pacific Grove, USA, 2004, pp. 100–114.
[4] A. Darvas, R. Hähnle, D. Sands, A theorem proving approach to analysis of secure information flow, in: D. Hutter, M. Ullmann (Eds.), Proc. 2nd International Conference on Security in Pervasive Computing, in: LNCS, vol. 3450, Springer-Verlag, 2005, pp. 193–209.
[5] R. Hähnle, J. Pan, P. Rümmer, D. Walter, Integration of a security type system into a program logic, Tech. Rep. 2007:1, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden (2007).
[6] S. Chong, A. C. Myers, K. Vikram, L. Zheng, Jif Reference Manual, Cornell University, version 3.0 edition, June 2006. http://www.cs.cornell.edu/jif/doc/jif-3.0.0/manual.html.
[7] K. Stenzel, Verification of JavaCard Programs, Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany, 2001.
[8] L. Burdy, A. Requet, J.-L. Lanet, Java applet correctness: A developer-oriented approach, in: Proc. Formal Methods Europe, Pisa, Italy, in: LNCS, vol. 2805, Springer-Verlag, 2003, pp. 422–439.
[9] B. Beckert, R. Hähnle, P. H. Schmitt (Eds.), Verification of Object-Oriented Software: The KeY Approach, in: LNCS, vol. 4334, Springer-Verlag, 2007.
[10] W. Mostowski, Formalisation and verification of Java Card security properties in dynamic logic, in: M. Cerioli (Ed.), Proc. Fundamental Approaches to Software Engineering (FASE), Edinburgh, in: LNCS, vol. 3442, Springer-Verlag, 2005, pp. 357–371.
[11] G.C. Necula, P. Lee, Safe, untrusted agents using proof-carrying code, in: G. Vigna (Ed.), Mobile Agents and Security, in: LNCS, vol. 1419, Springer-Verlag, 1998, pp. 61–91.
[12] MOBIUS Project Deliverable D 1.1, Resource and Information Flow Security Requirements, Mar. 2006.
[13] A.W. Appel, Foundational Proof-Carrying code, in: Proc. 16th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, Los Alamitos, CA, 2001, pp. 247–258.
[14] G.C. Necula, R.R. Schneck, A sound framework for untrustred verification-condition generators, in: Proc. IEEE Symposium on Logic in Computer Science LICS, Ottawa, Canada, IEEE Computer Society, 2003, pp. 248–260.
[15] A. Bernard, P. Lee, Temporal logic for proof-carrying code, in: A. Voronkov (Ed.), Proc. 18th International Conference on Automated Deduction CADE, Copenhagen, Denmark, in: Lecture Notes in Computer Science, vol. 2392, Springer-Verlag, 2002, pp. 31–46.
[16] S. Hunt, D. Sands, On flow-sensitive security types, in: J.G. Morrisett, S.L.P. Jones (Eds.), Symp. on Principles of Programming Languages (POPL), ACM Press, 2006, pp. 79–90.
[17] D. Volpano, G. Smith, C. Irvine, A sound type system for secure flow analysis, Journal of Computer Security 4 (3) (1996) 167–187.

[18] T. Amtoft, A. Banerjee, Information flow analysis in logical form, in: R. Giacobazzi (Ed.), 11th Static Analysis Symposium (SAS), Verona, Italy, in: LNCS, vol. 3148, Springer-Verlag, 2004, pp. 100–115.
[19] B. Beckert, A dynamic logic for the formal verification of Java Card programs, in: I. Attali, T. Jensen (Eds.), Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France, in: LNCS, vol. 2041, Springer-Verlag, 2001, pp. 6–24.
[20] D. Harel, D. Kozen, J. Tiuryn, Dynamic Logic, Foundations of Computing, MIT Press, 2000.
[21] P. Rümmer, Sequential, parallel, and quantified updates of first-order structures, in: Logic for Programming, Artificial Intelligence and Reasoning, in: LNCS, vol. 4246, Springer-Verlag, 2006, pp. 422–436.
[22] M.C. Fitting, First-Order Logic and Automated Theorem Proving, second ed., Springer-Verlag, New York, 1996.
[23] G.R. Andrews, R.P. Reitman, An axiomatic approach to information flow in programs, ACM Transactions on Programming Languages and Systems 2 (1) (1980) 56–76.
[24] L. Beringer, M. Hofmann, A. Momigliano, O. Shkaravska, Automatic certification of heap consumption, in: Logic for Programming, Artificial Intelligence, and Reasoning: 11th International Conference, LPAR 2004, Montevideo, Uruguay, vol. 3452, Springer-Verlag, 2005, pp. 347–362.
[25] L. Beringer, M. Hofmann, Secure information flow and program logics, CSF 00, 2007, 233–248.
[26] J. Pan, A theorem proving approach to analysis of secure information flow using data abstraction, Master's Thesis, Chalmers University of Technology, 2005.
[27] R. Focardi, R. Gorrieri, A classification of security properties for process algebras, Journal of Computer Security 3 (1) (1995) 5–33.
[28] P. Ryan, Mathematical models of computer security—tutorial lectures, in: R. Focardi, R. Gorrieri (Eds.), Foundations of Security Analysis and Design, in: LNCS, vol. 2171, Springer-Verlag, 2001, pp. 1–62.
[29] H. Mantel, Possibilistic definitions of security – An assembly kit –, in: Proc. IEEE Computer Security Foundations Workshop, 2000, pp. 185–199.
[30] A. Bossi, R. Focardi, C. Piazza, S. Rossi, Verifying persistent security properties, Computer Languages, Systems & Structures 30 (3–4) (2004) 231–258.
[31] A. Bossi, C. Piazza, S. Rossi, Compositional information flow security for concurrent programs, Journal of Computer Security 15 (3) (2007) 373–416.
[32] B. Beckert, M. Giese, R. Hähnle, V. Klebanov, P. Rümmer, S. Schlager, P.H. Schmitt, The KeY System 1.0 (deduction component), in: F. Pfenning (Ed.), Proc. 21st Conference on Automated Deduction (CADE), Bremen, Germany, in: LNCS, Springer-Verlag, 2007, pp. 379–384.